Implementing a Window System for an All Points Addressable Display

by

John Cambell Gonzalez

Submitted in Partial Fulfillment
of the Requirements for the

Degree of Bachelor of Science

at the

Massachusetts Institute of Technology

June, 1982

Signature of Author John Cambell Gonzalez
Department of Electrical Engineering and Computer Science, 1982

Certified by Richard E Zippel
Thesis Supervisor

Accepted by ______________________
Chairman, Departmental Committee on Theses

Implementing a Window System for an All Points Addressable Display

by

John Cambell Gonzalez, M.I.T. 1982

ABSTRACT

The design and implementation of a display management system for an all points addressable display is discussed. This window system exists as an independent software library, allowing application level programs to define arbitrarily overlapping rectangular areas of the screen. These areas, or *windows*, are used to view text, graphical entities, images, or any visual data form. Data to be displayed in these areas is represented in an abstract structure, the *canvas*, the form of which is defined by the application program.

The window system discussed in this paper was implemented on the Three Rivers PERQ personal computer. The performance of this implementation is examined and compared with the expectations of the original design.

Thesis Supervisor: Richard Zippel
Assistant Professor of Computer Science and Engineering

ACKNOWLEDGMENTS

The completion of this thesis would have been impossible without generous contributions of time and effort by a number of people. To list all who have assisted me would require more text than the body of this document. I would, however, like to mention a few.

To Richard Mark Soley, my tried and true friend, I would like to say thank you. Were it not for your advice and occasional wizardry, this thesis would not be.

I would like to thank Lynn Ilana Miller for her immeasurable commitment to the birthing of two theses. Her perseverance and patience are an inspiration to us all.

To Brewster L. Kahle, partner in all that befalls me, I would like to say a word of thanks. We got through.

I would like to thank Robert P. O'Hara for his enormous contribution to the design and implementation of *Jaws*.

I thank Ray Fessel, John Prager, Sheldon Borkin, and Peter Hardy for putting up with incessant questions and interuptions.

I thank Professor Richard Zippel, my thesis advisor and friend, for his great part in making this thesis a reality.

I thank Joy Hoppe, for her kind assistance in a time of need.

I thank Carl Hoffman for his good ideas and friendship.

For Cambell, Juanita, Amelia, and Anita

CONTENTS

FIGURES

Introduction 1

This thesis discusses research completed over the past year on a window management system (*Jaws*) for an all points addressable display. An implementation of the design outlined in this paper is currently underway on the Three Rivers PERQ minicomputer.

Jaws was designed to facilitate the handling of an all points addressable (APA) display device by application programs. Many software packages exist which will allow users to specify, manipulate, and output to rectangular areas of the screen. The thrust behind *Jaws* was to create a window system which provides these facilities while being transportable and independent of the underlying operating system.

Throughout this thesis male pronouns are used to denote the reader, programmers and general users. This is merely an arbitrary but well established convention of the English language. It is not intended to imply that all persons in the field of Computer Systems Engineering are men.

1.1 Clarification of Terminology

Before discussing window systems, it is necessary to define some terminology. These are not absolute definitions, but basics which will facilitate the review of this and other window system implementations.

All Points Addressable

Commonly abbreviated as *APA*. An all points addressable device is one in which every point is writable as an individual unit. An APA display has the capability of writing to individual *pixels* (c.f.). This differs from video character displays (e.g. the IBM 3270) which allow single characters to be written, but *not* the points which comprise those characters.

application program

A program written to perform some well specified action at the request of a user. Application programs rely upon underlying *system programs* to supply any system dependent software support they might need.

canvas

for this thesis, an area for communication between the window system and the application program. The canvas is in an application convenient data representation. It is similar in concept to the *world* of the Core Graphics package [Siggraph-ACM].

display

the physical device that the user sees. This will usually be a video terminal of some type.

keyboard

A device which allows the user to enter character data into the operating system or application program.

manager

A software subsystem which has been designed to perform a set of specific operations for a larger, usually enclosing, system. For example, a display manager might be responsible for transferring screen representations to the physical display device.

pixel

Sometimes called *pell.* A pixel is a single point on a raster scan display. In this thesis, a pixel is at times used to refer to a single bit in a bit plane which will later be moved to the physical display device. This is in recognition of the fact that although that bit is not currently on the screen, it represents a single point in an image which may appear on the screen.

process

An executing program *and* its associated data represent a *process*. Most modern operating systems are capable of suspending and later resuming the execution of processes without affecting the ultimate output of the program (unless the output is a function of real time).

screen representation

Loosely, a form of representing data which is understandable to the physical display device. For a raster-scan video display, this is usually pixels; for a vector display the screen representation is likely to be a list of vector endpoints; for a character matrix display (such as the IBM 3278), it might be a two dimensional character array.

terminal

A combination of a display, a keyboard, and perhaps a pointing device. Terminals are basic to user interaction in that they provide a means of getting information to and from the operating system.

window

A rectangular region of the display device. Windows may have ornaments (borders, titles, and the like), but the basic abstraction is that of a viewport through which data is examined.

1.2 Why Windows?

Information processing technology has progressed to the point that it is quite feasible, and in fact expected, for upcoming computer systems to support several concurrent processes. Windows were invented in response to the necessity for communicating with these processes as independent entities. It is quite useful to be able to reserve a section of the display device for the exclusive use of a particular program. This is not unlike the designation of certain areas of primary storage for an executing program. As with memory management, it became apparent that it is a rarity to fully utilize *all* of a specified resource (be that resource screen area or memory cells). The concept of partitioning the display area into regions, *windows*, each of which represents a communication pathway to an application program, sprung from this recognition.

There is no reason that a given application program needs to be limited to a single window; it may have several. Each may allow the program to express information in a different way (text, graphics, images, etc.). In a well designed program, the judicious use of windows greatly enhances the understandability, and hence the usability, of a software package. It is in this spirit that window systems are being actively researched.

1.3 Window Management Systems

Once committed to the idea of windows, one is faced with the challenge of how to implement them. *Window management systems* address this issue. Window systems usually provide facilities to: specify areas of the screen; output to those areas; manipulate (move, reshape, etc.) the windows; and to destroy them. Frequently, they provide routines which tailor a window to a specific application. These might include drawing characters, generating polygons, or panning over images. A common extension of this is to embed the window system in the overseeing application program. In these cases, the window system is really a part of that application package, and inaccessible to other programs which may wish to utilize it. Another extreme is to incorporate the window system in the underlying operating system. This forces the application programmer to use the system-wide display management facilities if he wishes to use any at all. *Jaws* was designed to be both non-embedded, and non-operating system dependent.

Overview of Selected Display Management Systems 2

This chapter provides a brief review of extant display management schemes. The merits and restrictions of each are discussed. The systems listed here in no way comprise an exhaustive list. Indeed, they represent a limited selection of the available strategies. The intent is to expose the reader to some of the more successful display management implementations so that he may have a context from which to view *Jaws.*

2.1 TSO Session Manager

The Time Sharing Option (TSO) Session Manager [McCrossin, O'Hara, and Koster] was designed to provide a facility to allow users to maintain a useful record of their transactions with the TSO environment while taking advantage of the added facilities of a video display terminal. The session manager was implemented as an additional layer insulating the user from the single line input and output mechanisms of TSO.

Most programs executing in the TSO environment operate in a traditional line at a time fashion. This is a reminder of the days when only character-printing teletypewriter terminals were available. While inelegant, the teletype did provide the user with a running record of his interaction with the computer system.

The advent of full screen character display terminals (such as the IBM 3270 series) generated some difficulties in the interactive computing arena. While these terminals frequently allowed smoother and faster user/program interaction, they also had the unfortunate property of not maintaining a log of those interactions. One of the design goals for the TSO session manager was to create a facility which at once provided the increased interactive capabilities of a display terminal, while retaining the journalizing features of a teletype.

Additional considerations included allowing the *user* to specify how he wishes to have program and operating system information presented to him, and designing the system such that minimal changes, if any, would be required in existing application and system programs.

The session manager channels its input and output through a mechanisms called *streams.* These streams are large character arrays which serve as buffers to hold incoming and outgoing terminal transactions. A window is defined to view a stream, and is updated when that stream is modified. An application program may both read from and write to a TSO stream. Because stream are able to retain much more information than the physical screen they are also used as a running journal of the interactive session. TSO streams are of particular interest because of their similarity the *Jaws* window system *canvas* (see section 5.2, below).

The primary display device for the TSO session manager is the IBM 3270 terminal. Windows may be specified as non-overlapping rectangles of the 3270 display. Due to the nature of the device, these window were designed to display only fixed width character data. At the time of its inception, the session manager was a definitive step towards more personalized computing. It is very good at what it was designed to do—manage a character display. Unfortunately, the commitment to character data and the limitation of not begin able to define overlapping windows make the TSO Session Manager obsolete when it is compared to modern display management systems.

2.2 Core Graphics Standard

The Core Graphics Standard [Siggraph-ACM] is an attempt to develop a standard definition for computer graphics packages. It is largely based on the definitions outlined by William Newman and Robert Sproull [Newman and Sproull]. The Standard was developed at a time when computer graphics systems were implemented almost exclusively on vector displays. It does not exploit the full capabilities of a raster scan device.

When using a graphics system based on the Core Standard, an application programmer must define an object referred to as the *world.* All of the information which will appear on the graphics device is part of this world. Windows are defined to have a *view* of the *world.* This view is made available on the physical device through a conceptual area known as a *viewport.* An unfortunate conflict of terminology is that these viewports correspond to the *windows* of *Jaws.* Viewports are rectangles on the physical screen which the user sees as containing information.

The Core Graphics Standards does not allow viewports to overlap. It is an interesting outline to study in that the concept of a single *world* of information is not unlike that of a canvas. Also, within the Core definition, windows are considered to *transform* the information they are viewing before displaying it. This is similar to the idea of *Jaws* windows translating information from an application program representation to a screen representation before transferring it to the physical screen.

2.3 Smalltalk Window System

The Smalltalk Window System [Tesler] was primarily implemented as a friendly user interface for the Smalltalk programming environment on the Xerox Alto personal computer. Smalltalk was formulated not as an operating system, but as an integrated programming environment. For this reason, it is at times difficult to separate the window system, from its implementation environment.

In the Smalltalk environment, windows are usually used to communicate with various system programs or *processes.* Some examples of a Smalltalk processes are the editor, the compiler, and the inspector. Each process passes its output to and receives its input from windows. These windows may overlap on the physical display, but each serves to identify a single executing program.

The major restriction of the Smalltalk window system is its strong association with a single programming environment. Within that environment, it is impossible to avoid the window system—outside of it the window system is unavailable.

2.4 New Lisp Machine Window System

The New Lisp Machine Window System [Weinreb and Moon] was designed to aid the user interface of the Lisp Machine [Lisp Manual]. While it facilitates communication with multiple processes, it is not the *only* means of process communication available to the user.

Using the Lisp Machine Window System (NLMWS), it is possible to define multiple windows any of which may overlap. A further convenience is that subwindows, or *panes* may be specified. A window may exist to contain the interaction between a user and one of many processes, or it may simply be the output area of any given process. These windows are capable of displaying graphics, text, and simple black and white images.

While much of *Jaws* was designed to emulate the NLMWS there are some aspects of that system which are undesirable. One of these is that a partially or completely overlapped window becomes inactive. That is, output to a window is suspended if another window encroaches on its screen area. A second problem is that to conserve primary storage, window images are generated once (by the application program) and stored only on the physical screen. If another process overwrites the display, the original image must be completely regenerated. This is usually rather difficult, and at times, impossible. A mechanism exists whereby the application programmer may request that a window to maintain an off-screen version of its display. This secondary area, known as a *screen buffer,* is automatically supplied for every window of the *Jaws* window system.

Design Objectives of Jaws 3

This chapter discusses the design objectives of *Jaws.* It is not concerned with the realization of the goals outlined, only their delineation.

3.1 The Type of Windows Displayed

Jaws is designed to manipulate rectangular windows up to the size of the physical display device. These windows may overlap, but cannot extend beyond the border of the screen.

Windows may themselves contain other windows called *children* or *child windows.* These must lie in the region bounded by the parent window. If the outer window moves, the child should also move with it, retaining its relative offset within the parent. When a child is created, it loses one window property: it may not arbitrarily overlap its siblings. This is because subwindows are usually created in an attempt to further subdivide a given section of the screen. That is, one would like a facility which allows groups of windows to be moved about as a unit. It is in this spirit that one creates child windows. To enforce this, it was decided that the design of *Jaws* should prohibit children from overlapping their siblings. If the programmer wishes to have independent windows which may be overlapped arbitrarily, he may create them as top-level windows.

Each window is a mechanism for viewing data generated by application programs. They may have borders, titles, margins, and numerous other frills. Conceptually, windows might be considered functions which translate information from user (application program) representations into screen images. To further understand this, it is necessary to consider how to establish the association between a window and the information it is to display.

3.2 Binding of Information to a Window

One of the major goals of *Jaws* was to retain a strong association between a window, and the data which are to be displayed in that window. It should never be the case that a window "forgets" what information it is displaying. This should be true regardless of what machinations occur to the physical screen (interrupts from other programs, error messages, etc.). The Lisp machine [Weinreb and Moon] keeps display information on the physical display—information which is lost if another process overwrites the display area. In such a system, the display must be regenerated by the application program (unless that window has an associated *screen buffer*). By comparison, the TSO Session Manager [McCrossin, O'Hara, and Koster] maintains the information needed to regenerate the display apart from the physical device. This seems the more useful and versatile approach in that a window's shape, position, and depth do not affect its contents. Any of the three parameters may be changed without having to recompute the screen image of the window.

A second consideration is that the information delivered to the window system should be in a representation convenient to the calling application program. While this is an unusual property of a display manager, it is not an unreasonable one to include. It would be quite useful for a graphic display window to accept vector endpoints, while a text window "understands" how to display linked lists of character strings. These considerations have been incorporated into the design of *Jaws*. Windows translate the information the application program passes them according to the *type* of the window (see section 6.2). With this mechanism, it has proved quite feasible to retain both the strong association desired, and a flexibility of information representation.

3.3 Dissociation of Display and Window Activity

In addition to divorcing the display storage from the window contents, one might consider having the update of a window be independent of its display status. Why not have partially, or even completely overlapped windows remain active? An answer might be to eliminate the useless computation involved in updating an image which is not visible. Conversely, the constant update of a partially overlapped window allows the system to display important messages and information which might otherwise be postponed. With *Jaws*, every window is updated at the request of the application program *regardless of its display status.* This means that not only partially overlapped, but completed covered and "hidden" (see *Window Depth Control*, section 6.2.2) will have their screen representations modified whenever the information they are viewing changes.

3.4 Non-Embedded Window System

The concept of a window system being *non-embedded* is somewhat difficult to convey. Such a system is one which exists above the functional level of the operating system. More simply, declaring a window system to be non-embedded indicates that an application program is not obligated to make use of it for visual (or other) communications. Indeed, it should be possible for many such window systems to coexist in one operating system. This does *not* imply that they should be able to operate concurrently, only that application programs should have a choice as to which one to make use of. *Jaws* was designed as a non-embedded system.

Neither should a window system be so application oriented that the application programmer must worry about the details of display management. The major advantage of a display management system is that it divorces the programmer for the specifics of handling the display. It would defeat this end if every application program attempted to implement its own window system. Another strike against individual application programs configuring their own window management systems is that there can be no hope of a uniform interface. *Jaws* is not the offshoot of some larger application program, but a software system in its own right.

In short, a *non-embedded* system is not tailored to any specific application program, nor to any particular operating system. It exists as a tool which, at the programmer's discretion, may or may not be part of the complete application package.

3.5 Flexibility

Of great importance to the design of *Jaws* is the desire to maintain flexibility. This includes flexibility in the implementation, flexibility in the display device, and flexibility in the application program's use of the window system.

Every attempt has been made to keep *Jaws* transportable from its initial implementation machine (Three Rivers PERQ, see chapter 7). *Jaws* is designed to drive devices of any kind, not merely visual displays, but interfaces which might include speech synthesis devices or Ethernet ports.

Finally, window systems are intended to make display management easy for application programmers. They should not unnecessarily restrict the programmer. Allowing multiple data representations is a step in this direction. The underlying window system should provide a well-defined interface which permits the application programmer to specify particular window attributes, without forcing him to do so when a default condition is adequate.

Design Overview 4

To facilitate its design and implementation, *Jaws* was divided into three functional parts: the *screen manager*, the *window manager*, and the *canvas manager.* In addition to these managers, the window system design depends heavily on the underlying data structures, and the inter-module communication protocol. These components are detailed in later sections. This chapter is intended to present a global view of how the individual pieces interact to achieve the objectives outlined in the preceding chapter.

4.1 An Overview of the Three Managers

The *screen manager* is responsible for handling the physical display device. It moves screen representations of window data to the display screen. It is the screen manager which negotiates overlaps among windows, for it is only when windows are displayed on the physical device that overlap becomes an issue. This manager may also clear the entire screen, or portions of it.

The *window manager*'s primary function is to convert information from application program representation (ASCII strings, vector endpoints, etc.) to screen representation (pixels). Since this may be done in a different way for each window type, the window manager must dispatch to type dependent procedures to perform a large part of the translation. In addition to this, the window manager controls the creation, destruction, and relative depth (in or out of the screen) of windows. The window manager does *not* concern itself with overlap. Each window is a separate entity which views information in an application convenient form (the *canvas*), and has a designated area in which to place the translated image (the *screen buffer*). Conversion from the canvas to the screen buffer may take place regardless of a window's display status. It is for this reason that the window manager does not have to examine the window overlap status.

The *canvas manager* handles the updating of the communication area common to both the application program and the window system—the *canvas.* The canvas contains information in a representation which is easy for the application program to understand. In some cases, it is desirable to isolate the application program from the low-level details of placing information in the canvas. The canvas manager may intervene in these cases. More importantly, the canvas manager notifies the window manager when a canvas has changed. The window manager must have a mechanism for determining when a canvas has changed, since a change in the canvas implies a change to the screen representation, and finally a change to the physical screen.

4.2 Intermanager Communication

During the operation of the window system, each manager is to perform a fairly specific portion of the job of getting the information onto the screen. To accomplish this, it is necessary to establish some sort of protocol by which the modules may communicate. For *Jaws*, queues of change records were chosen to facilitate the passing of information. Each record contains the range of influence of a change. Using these, each manager may determine how much recomputation, if any, is necessary to update the data structure it is responsible for.

It may prove instructional to trace the process by which the display is updated. The application program makes changes to the canvas, either directly or via calls to the canvas manager. In either case, the canvas manager must be notified of the changes, and given information as to their extent. This information is placed in the **changed-canvases queue**, and the window manager called.

Upon entry, the window manager examines the **changed-canvases queue** and determines which windows are affected by a change to the first canvas change record in the queue. For each of these windows, a type dependent translation procedure is called. This procedure translates the canvas information into pixels, placing the result in the window's *screen buffer*. The window manager then places a change record in the **redisplay queue**. The process is repeated for each canvas change record found in the **changed-canvases queue**, and then the screen manager is called.

When called, the screen manager examines the **redisplay queue** to determine which windows must be updated on the physical screen. For each of these windows, the screen manager moves the screen buffer onto the display, and restores any windows which are known to overlap that area. This is done for every window in the **redisplay queue**.

It is through these many levels of indirection that the display is updated. Usually, the queues will only hold the most recently changed item. The application program may make calls directly to the window manager if there is a need to create, destroy, move, or alter the depth of a window. The relationship between the major window system components is shown graphically in figure 1, below.

Above, the three managers are depicted as three synchronous processes, each beginning when another has finished. This need not be the case. Because each manager receives the information it needs from a global queue, there is no reason to prevent the managers from running concurrently. If this is the case, the calls from one manager to another should be eliminated, as each will begin processing when an item appears in its input queue.

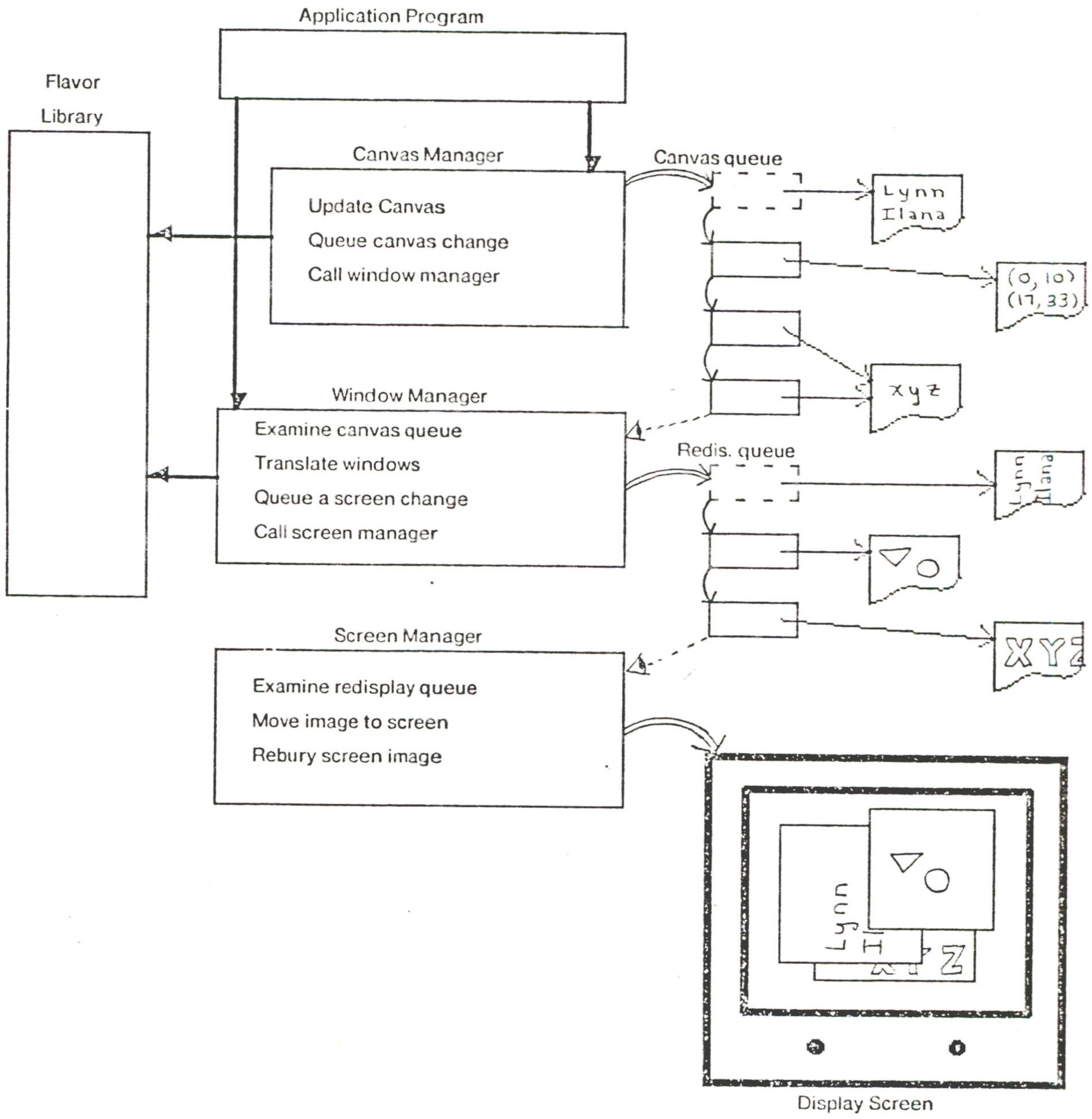
4.3 Dispatching by Window Type

To be able to translate from canvases of varying representations, it is necessary to have a mechanism for executing different translation code for each data format. In the Lisp Machine [Weinreb and Moon], this is done using the *flavor* facility [Lisp Manual]. For *Jaws* to be transportable, a different scheme had to be selected. Type dependent dispatches from generic procedures is one solution.

A type dependent dispatch means that there are some routines which examine an object's type field, and conditionally branch to code which will perform the operation for an object of that type. The result of the operation is conceptually the same for all types, yet the method for achieving that result varies. Such a dispatching procedure may be called a *generic* as it specifies an operation which will accept an argument of *any* type, and produce a correct result. Generic procedures have the advantage of being implementable in some form in almost any language on almost any operating system.

The most obvious place that a routine of this nature is needed is in the window translation procedure. Here, the window manager is to convert canvases of many types to screen buffers. Clearly this translation will be different for ASCII text and graphic orders. It is also necessary to place such generic procedures in the window creation, and destruction portions of the code. There are also times when the canvas manager must dispatch according to type (when allocating and deallocating the storage for a canvas, for instance).

Figure 1 - The Three Managers of Jaws



The relationship between the major components of the *Jaws* window system.

The body of the code executed once the branching decision has been made need not reside in the generic procedure, but may be located in another procedure or possibly even a different source module. If this is the case, it is possible to group all of the procedure bodies associated with a single type into a single module, and identify that module as a *flavor.* These are not to be confused with Lisp Machine flavors, except in the conceptual convenience of identifying a group of routines geared towards the handling of data in a particular representation.

4.4 Handling Input

An interesting question which comes to mind is whether or not the window system should handle input from the keyboard, mouse, or other transmission devices. The initial design of *Jaws* included provisions for such things. As the design progressed, it became apparent that *Jaws* was merely replicating the entry points native software provided for obtaining and buffering input data. In light of this, the design of *Jaws* was altered so as not to include input processing. Such input is handled by the application program, or in the *flavor* support packages called by the canvas manager.

The Data Structures of Jaws 5

The *Jaws* window system depends on its underlying data structures to hold the information needed for smooth operation. In addition to this function, some of the data structures described below serve as communication links between the various components of the window system.

5.1 Windows

By far the most central data structure to a window system is that of the window object. Windows must contain information indicating their size, their placement on the screen, their abstract contents, and how to translate the contents for the physical screen. In *Jaws*, some of this information is maintained not in the actual window record, but in associated structures (canvases and screen buffers). The depth of a window is determined from the interconnection of the window objects (see section 5.1.3 below). It is necessary to first get a feel for the basic window object before we explore the associated data constructs, and how each they come together to represent the window system information.

5.1.1 Basic Window Record

Below is the basic window data object. The functions of many of the fields are easily determined from their names. Others will need some clarification. The structure is:

Datatype: Window

Field Name	**Field Type**
windtype	*window_types*
prevwind, nextwind	*window_ptr*
parent, children	*window_ptr*
offsetx, offsety	*integer*
height, width	*integer*
topmargin, botmargin	*integer*
leftmargin, rightmargin	*integer*
canvas	*canvas_ptr*
viewposx, viewposy	*integer*
viewwidth, viewheight	*integer*
screen_buffer	*memory_ptr*
segment_number	*integer*
cursor_xpos, cusor_ypos	*integer*
mouse_xpos, mouse_ypos	*integer*
mouse_speed	*integer*
frills	*frill_ptr*

The window type slot is the most important field to the operation of the window system. It is this type field which determines how the information in the window is to be translated. Dispatching routines (see section 4.3, above) examine the window type to select what procedures to call to perform any type dependent processing. The value in window type slot identifies one of a set of defined window styles. This set is implemented using the Pascal enumerated scalar data type.

The *nextwind* and *prevwind* slots are used to give the window a position in a doubly-linked list. The *children* pointer, if non-null, points to the *first* child of a window. Parent fields are obvious. For a more complete explanation of these pointers, refer to section 5.1.3, below.

The *view* of a window declares what portion of the associated *canvas* this window is viewing. The canvas itself may be found at the end of the pointer in the *canvas* field of the window (see section 5.2, below).

A last thing to mention about the window record is the *segment_number* slot. This field holds the number of the memory segment assigned to hold the *screen buffer* of the window. Although it is an implementation level detail, this is mentioned because of its relevance to the next section.

5.1.2 Screen buffers

The *screen buffer* of a window is an area of storage designated to hold the screen representation of that window's data. They are similar to Lisp Machine [Weinreb and Moon] screen buffers, except that *every* window has a screen buffer. That is to say, every window maintains a copy of the pixels composing its display in off-screen memory.

The inclusion of screen buffers stems from a desire to update the screen as quickly as possible. With screen buffers, it is possible to perform the hard part of window update, the translation from user representation to screen representation, without slowing down the screen redisplay. Computation of the screen image may continue regardless of screen activity since the output of the translation is directed to non-displayed memory instead of to the screen. A further bonus is that repositioning windows becomes trivial—the screen manager need only change the location on the physical device to which it moves the screen buffer.

Using screen buffers, the job of the screen manager becomes conceptually trivial. All that need be done is move the screen buffers, or parts of them, to the physical screen *in the right order.* The order that the pre-computed screen buffers are moved to the display determines the apparent depth of the windows.

A variation on this theme occurs with child windows. A child shares its parent's screen buffer. This is an additional reason why children may not overlap. Since there is only one area of storage being used to maintain their images, if sibling windows were to overlap, their screen representations would also overlap. The result would be that the screen representations would not be independent, and operations such as subwindow repositioning would become rather difficult to perform. One might note that once a window is said to have children, it no longer makes sense to update the screen buffer of that window as a single object. If the *entire* screen buffer is updated, the screen representations of that window's children are destroyed, and must be regenerated from the canvas. A more meaningful operation is to update each child independently, modifying only a portion of the parent's screen buffer at a time.

A drawback to maintaining screen buffers is the large amount of storage required to maintain them. It may seem unreasonable to keep two copies of a screen image (one in memory, and one on the screen). More complete analysis shows that under usual circumstances, the memory sacrificed is not overwhelming. It is *extremely* unusual for the sum of the areas of all of the defined top level windows to be greater than the area of the display device. It is usually feasible to allocate enough memory to retain a single copy of the entire display screen. Maintaining screen buffers will be no worse.

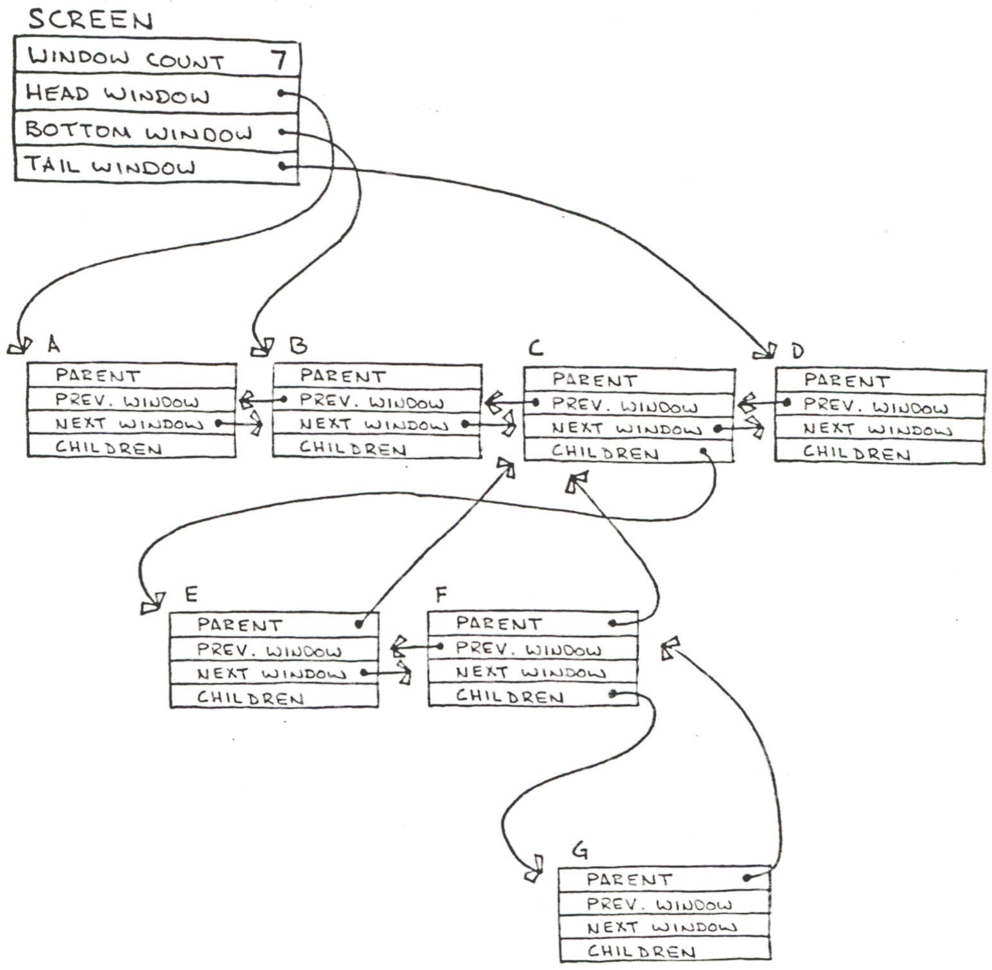
Screen buffers are implemented by assigning one PERQ memory segment to hold each. A memory segment is a group of memory cells which may be paged to and from disk as a single unit. This is an advantage, in that the screen buffers of seldom updated windows will gradually migrate to disk, and not consume valuable primary storage.

5.1.3 The Window Hierarchy

The window hierarchy is established using the relational pointers of individual window records. The form of the hierarchy is simple. Top level windows reside on a doubly-linked list, the head of which is held in the *screen* data structure (see section 5.4, below). If a window is to have children, a pointer to the first child created is placed in the *children* field of the parent window. All siblings of this first child are strung together on a doubly linked list rooted on that child. The *parent* field of each child points to the parent of the first child (hence the sibling relationship). Using this scheme, it is perfectly reasonable for children to in turn have children. A sample window record hierarchy is depicted graphically in Figure 2, following.

This linked list data structure is used for more than just storing the window records. It is used by the screen manager to determine the depth of a window. For each screen, there are three window pointers kept which together determine the display status of every other window. The first of these refers to a window called the *head window*. This is the first top-level window on the doubly-linked list. A second pointer is kept to identify the last window of the top-level linked list—the *tail window*. The last special pointer identifies the *bottom window*. The bottom window is the window which is to appear as the bottomost one of

Figure 2 - Sample Window Hierarchy



A sample web of window records representing four top level windows (only three of which are to be displayed); two child windows; and one grandchild. A possible screen configuration for this hierarchy is shown in Figure 3, below.

the display (i.e., the most buried). It is this window which is the first one moved to the screen by the screen manager. Any windows preceeding the bottom window on the linked list are not displayed. All windows following the bottom one appear on the screen as if they are stacked in the order determined by the window hierarchy. A diagram of a possible display for the hierarchy of Figure 2 is shown in Figure 3, below. The detailed use of the window hierarchy will be covered in section 6.1.

5.2 Canvases

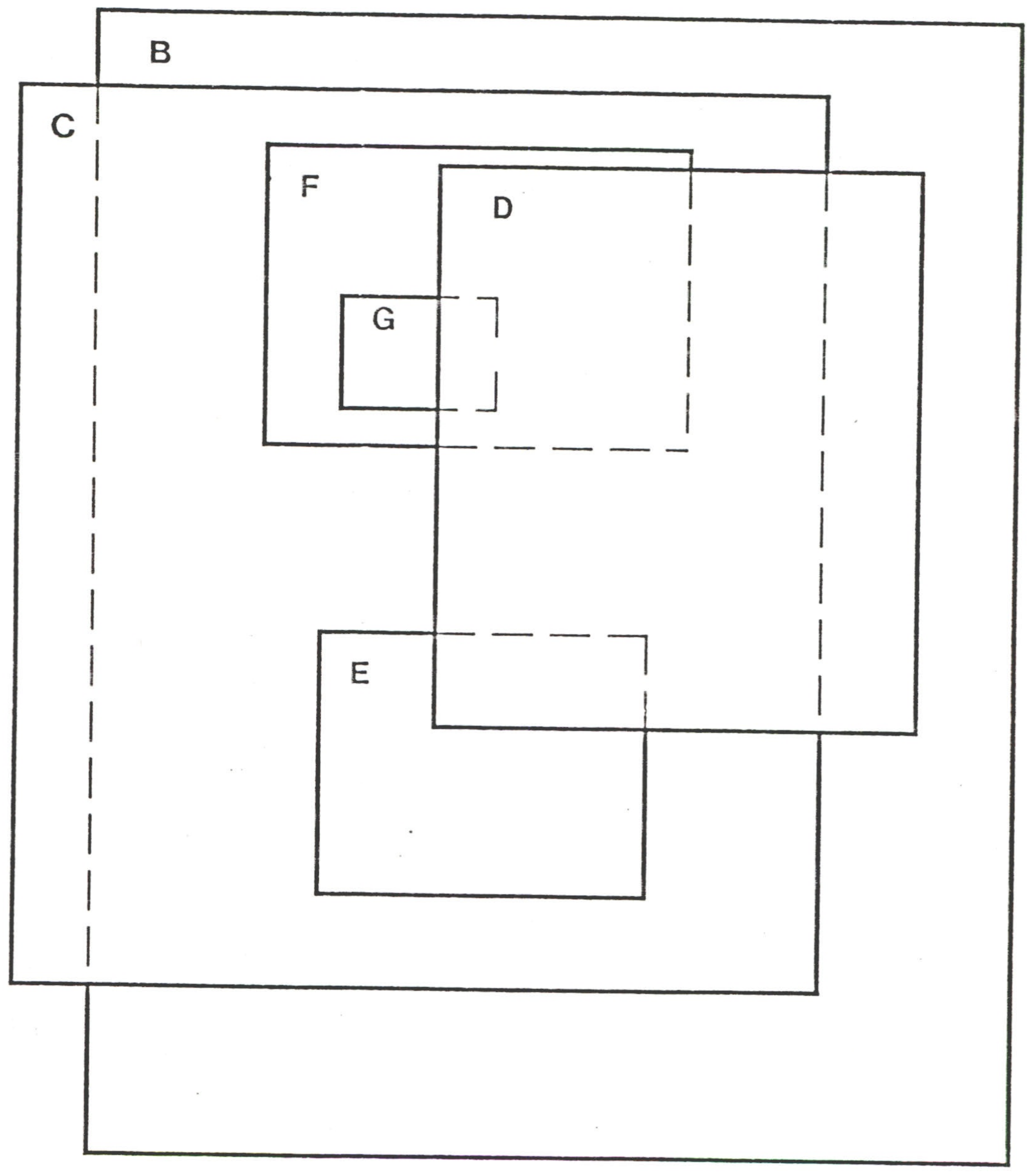
There has been a very strong emphasis on representing the contents of a window in a manner which is convenient to the application program making use of *Jaws.* The realization of this goal is accomplished via the canvas. It is difficult to specify the exact structure of a canvas, because that structure is defined by the specific application program. It is possible, however, to identify the purpose a canvas is to serve, and the basic information which must be contained in it.

The canvas is a communication area between the window system and the application program. As with the *world* of the Core Graphics Standard [Siggraph-ACM] and the *streams* of the TSO Session Manager [McCrossin, O'Hara, and Koster], it is here, in the canvas, that the application program makes modifications to the display information. It is the responsibility of the window system to notice.changes in the canvas, and to modify the screen display to reflect those changes.

Canvases are interpreted with the aid of routines found in type dependent support packages. These packages might be called *flavors.* A *flavor* is the combination of the window type, the canvas type, and the specialized procedures engineered to manipulate objects of that type. A good example of a canvas type is a text canvas. Within a text canvas, one might expect to find a linked list of ASCII strings. Similarly, an image canvas might be a simple bit plane. The window manager must be able to translate all of these window and canvas types. The method for translating the canvas data to a screen buffer is embodied in the flavor package for a window type.

Application programs are able to read from canvases as well. One example of a case where the application programmer may wish to retrieve data from a canvas is when it is necessary to identify what character the user is pointing to with the mouse. To be able to

Figure 3 - Overlapping Windows



A possible display configuration for the window hierarchy depicted in Figure 2.

determine this for a text window, there will need to be some mapping from a screen pixel to a canvas character. Only the canvas has the information needed to perform such a computation.

Despite their generality, there is some information which might be considered fundamental to all canvases. Before discussing this information, it is necessary to understand the concept of *canvas coordinates.* Canvas coordinates are simply integers whose interpretation depends on the type of a canvas. For instance, an integer in canvas coordinates for a text canvas might specify a certain number of characters, while for a graphic canvas, it represents an index into an array of vector endpoints.

There need not be a single window for any given canvas. Different windows may view the same canvas. A good example of this arises when a text canvas is created to hold a fairly large document. It would not be unreasonable to have two windows viewing different portions of this single document. In this manner, the user might compare two similar parts of the text and make corrections and additions based on his observations.

Because there is some information which should be maintained for every canvas, it was decided to head each conceptual canvas with a record containing basic canvas information. This is the *canvas record*:

Datatype: Canvas

Field Name	Field Type
canvas_type	*cantypes*
whose	*boolean*
width, height	*integer*
canvas_area	*POINTER*

The height and width are given in canvas coordinates. The *canvas_area* field holds a pointer to the actual storage area to which the application program will output to update the window. The *whose* field will be discussed with the *canvas manager* (section 6.3) below. It serves to indicate whether the canvas was created by the application program explicitly, or in some other way.

5.3 Queues

As mentioned above, the primary communication mechanism of *Jaws* is that of an information queue. Queues are used to transmit information between the canvas manager and the window manager, and between the window manager and the screen manager. Using this mechanism, one manager may notify another of a modification to the data area the two share.

A queue is an ordered collection of records. These records are accessed in a *first in first out* (*FIFO*) manner. That is to say that the least recently added record will be the first one examined by the reader of the queue. This ensures that sequential updates will occur in the order they were requested. The records themselves hold information specifying what portions of the common area have changed:

Datatype: Queue Record

<u>**Field Name**</u>	<u>**Field Type**</u>
next_record	*queue_rec_ptr*
queued_item	*POINTER*
modified_xpos, modified_ypos	*integer*
modified_width, modified_height	*integer*

The *queued_item* field of this record holds a pointer to the base of the area which has changed. If the queue involved is, say, the **redisplay queue**, then the *queued_item* slot will contain a pointer to a modified window. In this case, the integer fields would represent pixel offsets and values. If the **canvas queue** were under consideration, the queued item would be a pointer to a canvas record (see section 5.2, above) and the modified area would be defined in canvas coordinates.

These records are placed on a singly linked list. New records are added to the head of the list. This is known as *pushing* an item onto the queue. When a reader of the queue requires a record, the first record of the linked list is unlinked and returned. This is know as *popping* the queue. A queue itself is really nothing more than a header for the linked list of records:

Datatype: Queue

Field Name	Field Type
record_count	*integer*
record_list	*queue_rec_ptr*

Since the queue data structure is completely general, there may be, and in fact are, general routines for pushing and popping *any* queue.

5.4 Screens

One of the design objectives of *Jaws* was that it be capable of outputting to a variety of devices—not just a raster scan video terminal. The intelligence to output to other devices must be embodied in the screen manager. A screen data structure is needed to support that intelligence. In addition to this, screen records must retain the information necessary to handle window overlap, whatever that may mean for the device being controlled. At its present stage of development, *Jaws* is only able to control black and white video displays. The following data type was designed to support that particular style of device:

Datatype: Screen

Field Name	Field Type
screen_name	*string*
screen_address	*memory_ptr*
window_count	*integer*
head_window	*window_ptr*
bottom_window	*window_ptr*
tail_window	*window_ptr*

The *screen_address* field of this record holds a pointer to the actual location in memory that the operating system of the implementation machine recognizes as the screen. The *head_window* slot contains a pointer to the first top-level window of the window hierarchy.

The *bottom_window* pointer identifies that window which is to be the first one moved to the screen. Similarly, the *tail_window* field points to the last window of the top-level linked list structure. These three pointers are used by the screen manager to overlap the windows on the screen (see section 6.1, below). If a non-visible device were being controlled (perhaps a speech synthesis port), it is unclear whether or not the *bottom_window* field is useful. What does it mean to overlap things which can not be seen? This can only be determined on a per-screen basis.

A screen record is global to the window system. References are made to it by all three managers. This record, the **redisplay queue**, and the **canvas queue** are the only global variables of *Jaws.*

5.5 Window Frills

In addition to the basic window attributes (height, width, position, etc.), there are frequently special ornaments one would like a window to have. These might include such things as titles, borders, or grid lines. The information to generate such window ornaments resides in the *frills* slot of the window record. This field holds a pointer which is the head of a list of frills. Because frills vary greatly in how they are represented, each *frill record* on this linked list holds an untyped pointer to be used by whatever routine interprets and generates the ornament.

Datatype: Frill

Field Name	Field Type
frill_name	*string*
frill_info	*POINTER*
next_frill	*frill_ptr*

The Three Managers of Jaws 6

In its current implementation, *Jaws* is composed of three basic functional parts: the *screen manager*, the *window manager*, and the *canvas manager*. This chapter covers the details of each of these subsystems. Throughout the following discussion, there will be references to the data structures outlined in the preceding chapter.

6.1 The Screen Manager

The screen manager is responsible for moving the screen images of windows to the physical display device. These screen images may be found in the *screen buffers* of the windows. As windows may be defined to be overlapping, the screen manager must be capable of displaying images which appear to overlap. This is accomplished by simply moving the screen buffers to the screen in a bottom to top fashion. The overlapped portions of the lower windows will be overwritten by the later image placements. The result is similar to having individual pieces of paper which have been laid down on a table top.

The order to move the screen buffers to the screen is completely determined by the window hierarchy. Only the top level windows are involved in the update. Since child windows share their parent's screen buffer, there is no need to dive into the window hierarchy—the children will be updated when their parents are. The apparent effect of this is that all siblings appear to move as a single group. They are buried, surfaced, and repositioned as a unit. When there is an attempt to perform any of those actions to a child, the operation is redirected to the parent. Please refer to the window manager discussion (section 6.2), immediately below for further details on this aspect of the parent/child relationship.

To redisplay the screen, the screen manager first examines the **redisplay queue** to find a queue record referring to a window awaiting redisplay. A quick check is performed to ensure that this window is indeed on display. It then calculates the absolute offset of the modified area from the base of the physical screen area. The portion of the screen buffer the queue record specified as having changed is copied onto the screen. It is not always necessary to copy the full screen buffer to the screen. When a single character is added to a text window, it would be grossly inefficient to update the entire window. Instead, only a rectangle the size of that single character need be copied from the screen buffer to the display. The information delineating that sector may be found in the queue record which was popped from the **redisplay queue**.

Once the screen image of a window has been refreshed, the screen manager must *rebury* the updated image. To date, this reburying is performed in a rather unintelligent manner—the screen buffers of all windows following the redisplayed window in the window hierarchy are moved to the screen in the order they appear on the linked list of top-level window records. Using this algorithm, the image of a window which does not overlap *anything* will be copied from its screen buffer to the screen if *any* window preceding it in the window hierarchy is redisplayed. A more intelligent thing to do might be to only copy the screen buffers, or portions of the screen buffers, of those window which actually overlap the modified screen area. There is a very real tradeoff between the amount of time required to compute such overlaps, an the time needed to simply copy the bits of the image to the screen. For arbitrary window overlaps, the computation can be quite involved.

Occasionally, it is necessary to reconstruct the entire screen image from screen buffers. Such a situation arises when the application program is interrupted by another program which overwrites the screen. A full redisplay is accomplished by queueing the *bottom window* for redisplay. The reburying algorithm will cause every displayed window to be updated from its screen buffer.

After the first window on the **redisplay queue** has been processed in this manner, the next queue record is read and the process repeated. This continues until the **redisplay queue** is empty. In addition to moving screen images to the physical display, the screen manager includes entry points for clearing the screen area a window occupies. This must be done when a window is destroyed, buried, or moved (the screen area where the window *used to* be must be cleared before the window is repositioned).

6.2 The Window Manager

The window manager of *Jaws* is its largest and most characteristic module. This portion of the window system is responsible for creating, destroying, repositioning, and translating windows. The translation of a window involves converting the data that window is viewing in its associated canvas into pixels of the screen buffer. It is this function which is central to the window system.

There are additional services the window manager provides. These include examining the position and size of a window, reshaping it, and finding which window encloses a given point. The window manager is *not* expected to have to output to the physical display device—that is the screen manager's function. The window manager is to handle those aspects of the window system which require a full understanding of the window data abstraction, *and* its associated structures (screen buffers, canvases, frill lists and the like). The structure of many of these objects vary according to the *type* of the window involved. To better manage the processing of windows of varying types, type dependent dispatches are included in this and other sections of the window system.

6.2.1 Dispatching by Window Type

Many of the operations the window manager is to perform must be re-routed to procedure libraries tailored to manager windows of a particular type. These actions include creating, destroying, and translating windows. To create a window, the window manager first adds a blank record to the end of the window hierarchy. This record is filled in with the type, dimensions, and position of the new window. Once this has been done, a dispatch is made to the collection of routines to handle windows of that type. Here the type dependent initialization is done. When a window is created, there are often properties we would like to immediately add to the window. These vary with the type of the window. Such *frills* might include titles, borders, or margin size. Only after the type dependent parameters have been set is the new window queued for redisplay by the screen manager.

Similarly, a window of a given type may have "last requests" to be executed before the window is destroyed. A dispatch in the window destruction procedure provides such a facility. Only after the type dependent termination procedures have been executed does the window manager remove a window record from the window hierarchy.

By far the most important dispatch performed by the window manager is that for the translation of a window. It is impossible for the window manager to know how to interpret the canvas of every window type. It is for this reason that the actual translation is referred to the type library. In the procedure library for windows of a given type, there must be a procedure specifying how to translate the canvas into pixels. Once the translation has been performed, the window manager may queue the window for redisplay. It is interesting to note that the window manager cannot determine how much of the screen buffer has changed to optimize the screen manager redisplay. This information must be obtained from the procedure which actually translates the canvas because only the specific translation routines knows exactly how many bits of the screen buffer where changed.

6.2.2 Window Depth Control

The window manager is responsible for establishing the window depth. That is not to say that the window manager causes windows to appear overlapped on the physical display, only that it sets the depths the screen manager will use to update the screen. The screen manager refers to the window hierarchy to compute how windows overlap. Consequently, the window manager must alter that hierarchy if it receives a request to modify a window's relative depth.

There are three depth controlling operations an application program may specify. These are *surface, bury,* and *hide.* To surface a window is to bring it to the very top of the stack of windows. This is accomplished by simply moving the window to the end of the doubly-linked list. Burying a window means to place that window at the very bottom of the stack of displayed windows. This is easily done by moving the window record to a position of the linked list just preceding the *bottom window.* The *bottom window* pointer must then be set to point to this newly buried window. The effects of either of these two depth commands will become apparent during the next screen update.

The last depth operation is *hide.* When a window is hidden, it is not to be displayed at all. This does *not* imply that the window becomes inactive. That would violate the *Jaws* philosophy of separating the display status from the window activity. When a window is to be hidden, it is moved to the very beginning of the linked list. A check is then made to ensure that the *bottom window* pointer either points to a window further down the list, or is null. The screen area occupied by the newly hidden window is cleared and the screen manager is called to perform a complete redisplay. Since the now-hidden window does not appear after the *bottom window* its screen buffer will never be transferred to the display. The window disappears from the screen.

6.2.3 The Handling of Child Windows

One of the design objectives of *Jaws* is that it be able to handle windows with subwindows or *children.* Child windows are a way of further dividing the screen area. They may not overlap. A child is to move with its parent during repositioning and depth controlling operations. These windows are created in the same manner as top-level windows, except that the call for the creation of a child includes a non-null parent pointer. The new window is added to the hierarchy as a child of that parent window. Offsets are computed as offsets from the origin of the parent. When the parent is destroyed, all of its children, indeed all of its descendant, must also be terminated.

It is difficult to define a consistent behavior for children during depth control operations. Clearly, when a top level window is buried, its descendants should be buried with it. What is to happen if a *child* is buried? Since children exists only under their parent's auspices, the child should be buried within the frame of the parent. This would be a useless operation because *children may not overlap.* Burying a non-overlapping window is meaningless. A similar problem arises with the hiding of child windows. Should the child completely disappear from the display? The convention finally decided upon was that any depth controlling operation performed on a child is redirected to the parent. Thus to burying a child implies that the child's siblings, and their common parent, will all be buried as a unit. The reasoning is that a subwindow is inextricably bound to its parent from creation. What happens to a child should also happen to its siblings and the parent. The only exception to this convention is window movement. Moving a child window means moving it relative to its parent.

The last peculiarity of child windows is the fact that they share their parent's screen buffer. The implications of this are not all immediately apparent. When the screen image of the parent is repositioned, so are those of its children. Subwindows may be updated freely, but that update takes place in the screen buffer of the parent window. Consequently, if the parent is then updated, the screen image of its child window is lost. In a sense, parents only serve to bind their children together. Once a window is declared to have subwindows, it loses its ability to operate as a full viewport, but instead becomes a *frame* which holds other, smaller, viewports (the subwindows). These are not unlike the *frames* of the Lisp Machine Window System [Weinreb and Moon].

6.2.4 Other Window Manager Functions

In addition to the basic function of creating, destroying, updating, and setting the depth of windows, the window manager provides a few other services. These include repositioning, querying, and locating window objects. Repositioning a window is very straightforward—the *offset* fields of the window record concerned is modified to reflect the requested movement. The only checks performed are those to ensure that the window remains within a bounded area (the parent's borders for a child; the screen border for a top-level window). Querying a window returns the window's position, size, and type.

An application program may need to determine what window contains a given screen point. One example is when the user is selecting windows with the mouse or other pointing device. The window manager is able to provide this information. To find the window which contains a specific point, the window manager searches the window hierarchy beginning with the *tail window*. A backward search is appropriate because the window hierarchy establishes the depth of each window on the screen. What must be returned is the uppermost window which contains the point, *not* any window which might be underneath that one. Once a window is found to contain the point, any children that window might have must in turn be searched. A recursive search is necessary because the most specific answer, i.e. the smallest enclosing window, is what is desired.

6.3 The Canvas Manager

The canvas manager of *Jaws* is the module responsible for the creation, destruction, and, in some cases, the modification of canvases. As the specific structure of each canvas is *very* application program dependent, the canvas manager frequently dispatches to procedure libraries for handling canvases of a particular type.

When specifying that a window is to be created, the application programmer must decide whether the application program or the canvas manager will be responsible for maintaining the canvas. There are examples for either situation. If the application programmer wishes the window to behave as a simple typewriter, then the canvas manager may easily handle placing characters in the canvas. If, on the other hand, the program is an image processor, the application program may need to modify individual pixels in the canvas. In this case, the application program may "own" the canvas, and be responsible for its update. The ownership of a canvas may be determined by examining the *whose* field of the canvas record.

Regardless of who owns it, the canvas manager must be notified of any changes to the canvas. This notification includes information delineating what area of the canvas has changed. If several changes are to be made before a screen update, the canvas manager keeps track of the accumulated modifications, and queues them all at once. This facility prevents the application program from having to designate a series of local canvas changes. Instead, the program may modify separate areas of the canvas, and depend on the canvas manager to keep a cumulative account of the rectangle encompassing all modifications. This rectangle represents the screen area which should be refreshed during the next screen update. This information is placed in a queue record, to be acted upon by the window manager when a refresh request is next signalled.

The canvas manager is not aware of which windows, if any, are viewing the canvases it is maintaining. It exists primarily to insulate the application programmer from some of the lower level details of the window system. With the canvas manager, an application program may make use of an output area without concerning itself with display restrictions, such as window size, position, or visibility. The canvas retains its state regardless of any of these factors.

The canvas manager may be considered as an application program in its own right, since it only manipulates the canvases, or *communication area,* of the window system. It is the existence of the canvas manager that allows a library of generally useful window operations to be defined. Echoing characters from the keyboard is something the canvas manager might do. To echo a character, the canvas of a window must have that character added to it. Only after the character appears in the canvas will it make its way to the screen. By introducing this use of the canvas manager, the application programmer need not be concerned with echoing, or even *reading*, characters. Input characters may be extracted from the canvas, instead of the system input buffer.

The Implementation of Jaws 7

An implementation of the *Jaws* window system was begun during January, 1982 on the Three Rivers PERQ personal computer. Since that time, a working version has been successfully coded and tested. This chapter describes some of the relevant details of that implementation.

7.1 The Pascal Programming Language

Jaws is implemented in the Pascal programming language. While not the most powerful computer language available on today's machines, it does provide a good selection of programming features. The PERQ implementation of Pascal includes a several language enhancements which proved quite useful during the coding of the window system (see section 7.2, immediately below). Pascal is also the only language currently available on the PERQ.

Pascal does not permit the storage of function objects in a compound data structure. This restriction made implementing the window type dependent dispatches somewhat more difficult than it might have been is a more object oriented language such as Lisp [Lisp Manual]. In Lisp, the function to be executed by a particular window to perform a generic action might be stored in the window record itself. A different mechanism had to be found which could be realized in Pascal. Here, generic procedures are called to dispatch to a specific routine for handling a window of a given type. Since the PERQ implementation represents the type of a window as an enumerated scalar, it is quite feasible to use the *wind_type* field in a simple ***CASE*** statement which selects a procedure to perform the actual operation. An unfortunate consequence of this scheme is that when a new window type is added to the window system, this dispatch module, indeed *all* of the window system modules must be recompiled.

7.2 The PERQ Personal Computer

The Three Rivers PERQ mini-computer is a single user system quite suited to a programming project of this nature. It supports a black and white, high density, bit mapped display and a tablet which allows the entry planar coordinates into the operating system.

7.2.1 PERQ Software Enhancements

In addition to its hardware features, the PERQ provides many software enhancements which greatly facilitated the implementation of *Jaws.* The first of these is a Pascal intrinsic called RasterOp. RasterOp allows a programmer to move and logically combine very large areas of storage. Its primary purpose is to move screen images to and from the PERQ's display screen. Using RasterOp, orders for the movement of entire screen buffers were reduced to a single Pascal statement. As RasterOp is implemented in microcode, it is a *very* fast operation. This proved to be crucial for real time window system applications.

Another PERQ Pascal addition is that of supporting generic pointer types. While the window system could have been implemented using the standard definition of Pascal [Jensen and Wirth], generic types allowed general data structures such as canvases to be included in record structures. Where there no generic types, these data types would have to be implemented as variant records.

The PERQ also features a software module import/export facility. Using this, it was possible to divide *Jaws* into many modules which could be independently modified and recompiled. Please refer to the following section for a list of the modules which comprise the window system.

7.3 The Modules of Jaws

Jaws is a large software system. It was neither feasible nor desirable to implement a system of its extent in only three independent software modules. There are many. This section identifies those packages and provides a brief description of the function of each:

window datatype declarations

This module contains the declarations of the window system data types. When this package is modified, *every other module of the window system must be recompiled.* This is because *every* other module must refer to the window system data structures to perform its function.

window system utilities

The window system utility package includes those routines which are frequently referenced by the other system modules. In this package are functions which: return a window's eldest ancestor; compute a window's absolute offset from the origin of the physical screen; push a queue record onto any specified queue; etc.

screen manager

The screen manager (see section 6.1, above) is responsible for updating the physical display device. This module contains the screen manager procedures.

window manager

The window manager creates, destroys, repositions, and translates windows. It is fully described in section 6.2, above. This module contains the window manager procedures.

canvas manager

This module contains the canvas manager routines. The canvas manager handles the creation, destruction, and manipulation of canvases. It does *not* know how to translate canvases into screen buffers. Refer to section 6.3, above.

window system dispatcher

This module contains *all* of the dispatch routines of the window system. Every "generic procedure" may be found in this module. The reasoning behind placing all such routines in a single package is that only this file and the window data type declaration module need be modified when a new "flavor" is added to the window system. The window types module must be updated to reflect the additional window type.

basic flavor package

The basic flavor package is a module which contains routines to perform functions frequently used in flavor packages. Some of the basic flavor package procedures: draw borders around windows, redefine window margins, alter the view a window has of the canvas, and clear a window's *screen buffer* (as opposed to its screen area).

bit window flavor

The bit window flavor package was the first "flavor" of window implemented. It supports simple bit mapped windows. This package includes the target of the dispatches for type *bitmap*. It is here that the body of the routines for translating the canvas of a bit window to the screen buffer reside. Since the canvas is also a bit plane, that translation is a simple copy operation.

Currently, there are two other flavor packages implemented for *Jaws*. These are the *character array* and *Quix* flavors. Ultimately, there will be many more. *Jaws* was designed to allow the application programmer to define and display whatever data abstraction he finds convenient. It was towards this end, that flavor packages were designed. The unfortunate aspects of *Jaws* flavors are that they require the designer of the flavor know a fair amount about the internals of the window system, and that every flavor addition forces a recompilation of the entire window system.

Future Directions for the Window System 8

The PERQ implementation of *Jaws* both proved its design features, and revealed its design oversights. This chapter discusses the results of the PERQ implementation, and briefly reviews some enhancements and omissions which may improve the performance of this and other window management systems.

Overall, the performance of the *Jaws* implementation is far better than anticipated. It was expected that due to the overhead of the communication queues and copying screen images to the screen, the window system would not be capable of keeping up with real time application programs. This is not the case. *Jaws* is able to keep pace with interactive application programs. This includes such things as typing into character windows, and dragging windows of any type with the mouse.

Likewise, the redisplay algorithms and canvas scheme worked quite well. Canvases proved a *very* convenient way of having two windows display the same thing. This was as expected, but the visual affect is much more impressive than imagined. An analysis of the storage consumption was not performed, yet no application program to date has run against a storage limitation.

The most apparent misfeature of *Jaws* is the overlap management. When partially buried windows are updated very frequently (about three times a second), the flashing of the screen as it struggles to rebury the window after update becomes noticeable. At Five updates a second, the flashing is annoying, while a window update rate greater than eight per second is intolerable. A possible solution to this problem is proposed in section 8.1.1, below.

The major design flaw is probably that of choosing to have windows share their parent's screen buffer. This proved to be a rather troublesome property. Children must be handled specially by almost every routine of the window manager. Many of these difficulties are compounded by the fact that every screen buffer is contained in a single segment. Since each segment is a distinct memory object, it is difficult to reference a smaller rectangle defined to lie within that segment. A child window is just that. The alternatives, are either to have a screen buffer for every window buffer, or to have the inclusion of screen buffer a per window option selected by the application programmer. Both of these are rather unattractive.

8.1 Possible Improvements to Jaws

Jaws is by no means perfect. There are many design alterations which would greatly contribute to a more robust and general system. The major design changes have been mentioned above. Following are possible additions or modifications to improve the PERQ implementation of *Jaws*. Most of the enhancements outlined in this section should also be incorporated in any redesign of the window system.

8.1.1 Improving the Overlap Algorithm

It was mentioned that the flashing of the screen as overlapped windows are redisplayed proved distracting. One way to solve this problem is to add to the screen manager the intelligence to compute when and to what extent windows overlap. With a smarter screen manager, only the visible areas of partially covered windows would be refreshed. A completely covered window would not be updated at all. Because only the visible portions of window are shown, there would no longer be a need *rebury* window images. It is the rebury operation which is the source of the screen flash. Consequently, this flash would be eliminated from the revised system. A second method of eliminating the overlap is to retain the original screen manager, and to instead add an specialized *overlap* manager.

8.1.2 Adding Other Managers

The global queue method of information transfer used in *Jaws* facilitates the insertion of other functional managers between any of the existing ones. A reasonable addition might be a *display overlap manager.* This module would compute what portions of windows which are to be redisplayed are actually visible. These visible sectors will always be decomposable into some combination of rectangles. The overlap manager would inspect the window hierarchy, and the **redisplay queue** and use that information to create a new queue of the smaller rectangles which together form the visible display image. The screen manager is already set up to display any subrectangle of given window. There would be no need to modify its behavior other than to designate this new queue as its input stream.

Another manager to add to the system would be one to handle input to the window system. Currently input is either handled by the canvas manager, or by the application program itself. An input manager would poll all of the input devices and direct the input streams to windows designated as *active*. The canvas manager would then take over, updating the canvases of the active windows, and notifying the application programs managing those windows that they have input. Before such a scheme may become a reality, the underlying operating system must provide some means of managing independently executing processes. The continuous inspection of input buffers requires an asynchronous process.

8.1.3 Asynchonous Managers

There is no good reason the three main managers of *Jaws* cannot run as asynchronous processes. Currently, one manager calls the next in a well defined sequence. This does not always make the best use of system resources. It may well be more efficient to have the window manager active recomputing the screen image of a complex data representation while the user is performing a relatively less demanding activity such as pure text entry.

There are some drawbacks to parallel processes. The first problem is that the PERQ does not currently support simultaneously executing programs. This feature is to be added in the near future. Once it becomes available, there are still difficulties. A locking mechanism will be needed to ensure that no area is being read and modified simultaneously. The results of such an action are quite unpredictable. Once all of the problems have been resolved, it is expected that multiprocessing will greatly enhance the performance of *Jaws.*

8.1.4 Multiple Screens

Usually, the user is only interested in managing a single screen of windows. One can imagine situations where this assumption is no longer valid. An example is if the operating system supports a color screen in addition to its usual alphanumeric one. The window management system should be able to control many screens if that is what the application programmer desires.

The entire state of the window system is embodied in the screen record and the two information queues. Currently these the are *only* global variable of *Jaws.* It is quite feasible to create an additional data type to hold all such global information on a linked list or similar structure. The global variables could then be re-instantiated every time the application requests a screen change. In this manner, a single invocation of the window system might manage multiple screens of varying types.

A question arises as to what other modifications would need to be made to *Jaws* to handle screens of other types. The largest alterations would occur in the routines which handle the screen representations or *screen buffers.* These must be made to understand whatever screen representation is appropriate for the currently active screen. Fortunately, all such routines reside only in the screen manager and the translation portions of the window manager.

Conclusion 9

As personal and main frame computers supporting all points addressable displays become more prolific, it will prove increasingly necessary to supply the users of these systems with reliable and well designed display management facilities. The *Jaws* window management system is a step in that direction. *Jaws* is by no means the best window manager in existence. It is an attempt to incorporate the current display management ideas the author feels are the most useful into a coherent and flexible display driver.

The *Jaws* implementation employs some software concepts seldom found in window systems of this nature. Among these are the use of communication queues to relay redisplay information, a facility which allows application programmers to express information in an *application convenient* representation, and the use of the window data structures to store both the data for individual windows, and the information needed to establish the relationships between displayed window images.

As an experimental system, *Jaws* was quite successful. It performed better than expected. Design problems were uncovered, but that is part of motivation behind any prototypical system—to find the limitations of the design and implementation. The author learned much both about window systems, and implementing large software subsystems. It is hoped that the development of *Jaws* will continue, and that this initial design and implementation effort will not have been in vain.

References 10

[Jensen and Wirth]

Jensen, Kathleen, and Wirth, Niklaus, PASCAL User Manual and Report, Springer-Verlag, New York, NY., 1978

[Lisp Manual]

Weinreb, Daniel L., and Moon, David, The Lisp Machine Manual, M.I.T. Artificial Intelligence Laboratory, Cambridge, MA, July 1981.

[McCrossin, O'Hara, and Koster]

McCrossin, J.M., O'Hara, R.P., and Koster, L.R., "A Time-Sharing Display Terminal Session Manager", IBM Systems Journal, vol. 17, num. 3, IBM Corp., 1978

[Newman and Sproull]

Newman, William M., and Sproull, Robert E.; Principles of Interactive Computer Graphics, McGraw-Hill, New York, NY, 1979

[Siggraph-ACM]

Computer Graphics, A Quarterly Report of SIGGRAPH-ACM, Association for Computing Machinery, New York, NY; August, 1979

[SPICE]

Ball, J. Eugene, "Alto as Terminal", Carnegie-Mellon University SPICE Project, Spice Document S008, Carnegie Mellon University, March, 1980

[Tesler]

Tesler, Larry, "The Smalltalk Environment" Byte, Byte Publications Inc., August, 1981, vol. 6 num. 8

[Virtual Terminal]

Lantz, Keith A., and Rashid, Richard F., "Virtual Terminal Management in a Multiple Process Environment" Proceedings of the Seventh Symposium on Operating System Principles Association for Computing Machinery, Pacific Grove, California, December, 1979

[Weinreb and Moon]

Weinreb, Daniel, and Moon, David A., Introduction to Using the Window System, M.I.T. Artificial Intelligence Laboratory Working Paper 210, May, 1981.